TimbreWorks Embedded Corp

Embedded Timbre

This document covers Embedded Timbre and its use in an embedded environment

Embedded Timbre (ET) by TimbreWorks is licensed under a Creative Commons Attribution 3.0 Unported License.

Synopsis 2

Overview 2

Queues 3

API for Byte Queues 4

API for Cell Queues 4

Timeouts 4

API 4

Machines 5

Time Machine 6

State Machine 6

API 7

CLI 7

List of words 8

Porting ET 10

Development 11

Synopsis

Embedded Timbre (ET) is an embedded bare metal operating system, development environment and debugger with a command line interface (CLI) that promotes interactive development and testing on the target device. Development takes place by interacting with the embedded program over a serial stream which can be embedded in a protocol such as SFP to allow multiple services to be available including firmware updates. A host program, TimbreTalk, running on Windows, Mac or Linux, is used to run the other end of the SFP link and provide an enhanced command line interface. Commands are typed in on the host, executed on the target and the output is displayed on the host.

ET resources include: queues, timeouts, machines and an extendable CLI.

Overview

Queues are a fundamental form of encapsulated data flow. They allow isolation between a source and a sink as well as elasticity and accumulation. In a system, data is always moving through in one form or another being transformed along the way. A queue represents one way of containing that data or temporarily storing a segment of data flow. At the simplest level, individual data elements in the queue are fundamental data types such as bytes or integers. More complex objects are queued as pointers. Queues aid in the factoring of a data flow into its constituent parts and give data a space to live in for a finite time.

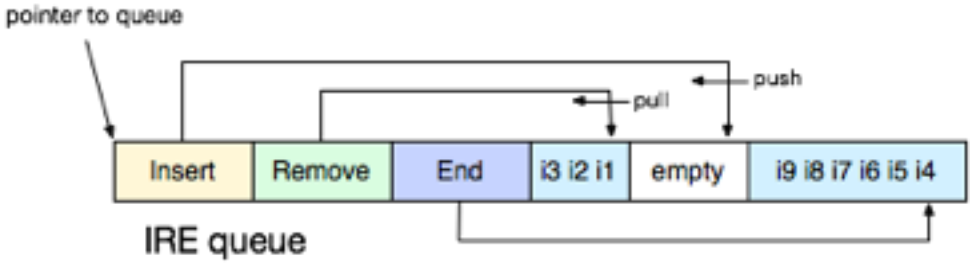
Timeouts offer a way for bookending results in real time. Real time operations are driven by events. Sometimes these events do not occur or do not occur within the appropriate time interval. This is where timeouts can be used as a way of managing a process and keeping to a schedule of events in real time. They can also be used to sample or react periodically like blinking a status light.

Machines are the processing elements which work on the data or just respond to events within the system. By their nature machines are simply code which runs to completion in a finite amount of time without hogging the CPU. It is based on a programming philosophy which embodies simplicity and embraces transparency of operation. A machine is simply a function which takes no arguments and leaves no results. At its simplest, a machine will run once and be done. More complex machines can be build by connecting simpler ones together and making use of queues and timeouts.

The above resource can be combined to create state machines, time machines, data handlers and so on. One example of a more complex machine is the command line interface or CLI. The CLI uses queues to manages data flow from and to a source and sink such as a serial port. It uses machines to process the input and generate output. Timeouts are used to make it reliable. Part of the CLI machine processing involves gathering input characters into words and then processing them as commands, numbers and strings within a context provided by the developer. The developer builds on the basic system and customizes to the environment they are working in to facilitate interactive development, aid in debugging and to provide an environment which can be engaged in real time while the system is in operation. This allows state changes, viewing of internal operations and listing of operational statistics. Additionally, a macro facility exists where additional commands can be built up at run time to further enhance the developer's productivity.

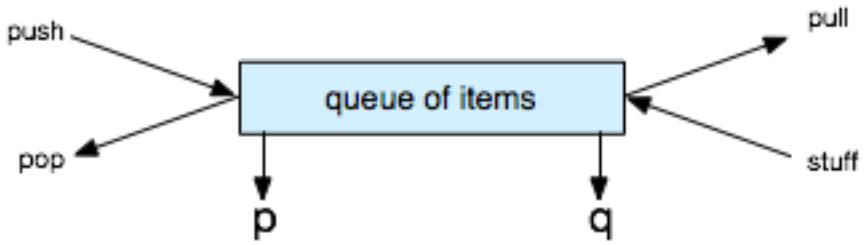
Queues

Two types of queues are available in ET. Byte queues and Cell queues. Byte queues are compact and are for capturing short byte data flows efficiently. Cell queues are for handling everything else and can be as large as needed. Both queues are based on the IRE (insert, remove, end) queue data structure. Queues are fixed length at the time of instantiation where that be at compile time or run time. Separate pointers are used for managing inflow and outflow allowing the data structure to be used by separate processes without the need for locking or blocking. The end pointer is used to make the queue circular in memory to make it safe and impossible to overwrite other memory. Part of the API provides information on the state of the queue so that its fullness or emptiness can be managed.



IRE queue structure features:

- size is fixed at declaration
- three pointers followed by an array of items
- insert and remove pointers move towards the end pointer when they equal the end pointer location, they are replaced with the end pointer, starting at the other end of the items array making it circular
- insert pointer always points to an empty location to push while remove pointer points to the item to pull
- dual asynchronous access
- since the insert and removal pointers are separate, two processes can access the queue at the same time no blocking between pulling and pushing is needed
- useful for queueing data from an interrupt to a different process running at a different level of priority
- queue with 1 item can use pointers as semaphores if needed to manage other resources



Since the structure is quite simple, it is easy to add additional functions allowing double ended operations like doing and undoing. A stack can be created with the push and pop functions. Queues can be reversed by making use of the stuff function.

API for Byte Queues

```
BQUEUE(size, name) - for declaring a byte queue; preinitialized
NEW_BQ(size, name) - for declaring as part of a structure; must be use INIT_BQ
INIT_BQ(bq) - for initializing a byte queue structure at run time
void zerobq(Byte *q) - empty out the queue
Byte bq(Byte *q) - get copy of first item
void pushbq(Byte b, Byte *q) - push item into queue
Byte pullbq(Byte *q) - pull item from queue
Byte qbq(Byte *q) - get number of items in queue
Byte sizebq(Byte *q) - get number of items queue can hold
Byte leftbq(q) - how much is left

Byte fullbq(Byte *q) - indicate if queue is full
```

API for Cell Queues

Similar to byte queue but with a few additions and one deletion:

```
QUEUE(size, name) - for declaring a queue; preinitialized
NEW_Q(size, name) - for declaring as part of a structure; must be use INIT_Q
INIT_Q(q) - for initializing a byte queue structure at run time
void zeroq(Cell *) - empty the queue
Cell q(Cell *) - copy of first item at head of queue
void pushq(Cell , Cell *) - push item into queue
Cell pullq(Cell *) - pull item from queue
Cell queryq(Cell *) - get number of items in queue
Cell sizeq(Cell *) - get maximum number of items queue can hold
Cell leftq(Cell *) - how many more items will fit in queue

Cell p(Cell *) - copy of last item at end of queue
Cell popq(Cell *) - pop item from queue
void stuffq(Cell , Cell *) - stuff item into queue
void rotateq(Cell *, Cell n) - rotate n queue items
void transferq(Cell *, Cell *, Cell n) - transfer n items between queues
```

Timeouts

An efficient resource for noting time intervals is called Timeouts. With this tool, the developer declares a timeout structure, sets it to a time in the future, and then checks to see when the timeout occurs. There is a one interrupt timer which counts milliseconds. The timeouts use this time base and set a time in the future. When that time arrives, the timeout will be over. On a 32 bit system timeouts up to 49 days with a millisecond resolution are allowed.

API

```
TO_MILLISECONDS
TO_SECONDS
TO_MINUTES
```

```
TO_HOURS
TO_DAYS
bool checkTimeout(Timeout *timer) - see if it has timed out
void setTimeout(Long time, Timeout *timer) - set and turn on the timeout
Integer sinceTimeout(Timeout *timer) - get time since last timeout
void stopTimeout(Timeout *timer) - stop a timeout
void startTimeout(Timeout *timer) - turn on a timeout
void repeatTimeout(Timeout *timer) - advance the timeout time and start it
void timeoutWait(Cell time) - timed delay loop
```

Machines

A machine is a C function which performs a task that has no arguments, no return values and no forever loop. It is a function which will run to completion in a reasonable time. For most systems that would be less than a millisecond. For functions that need to run immediately, there are interrupts and their life is measured in microseconds. ET keeps the machines in a queue. When each one is done, the next one in the queue is run. This way the machines are run atomically which makes sharing resources much easier.

There is an machines API for queueing machines. Each machine must reinsert itself into the queue if it needs to run again or it could insert other machines into the queue.

Here's a simple example of a machine that services the SFP nodes:

```
void serviceNodes(void)
{
  int i;

  for (i=0; i<maxnodes; i++)
       if (nodeDown[i] == 0)
            serviceNode(nodes[i]);
  activate(serviceNodes);
}
```

This machine will run over and over again since it queues itself up with the activate() construct. In an RTOS system which supported task switching a different way to code this would be to make it a loop. This adds complexity to the system to suspend loops, manage context and supervise the run time of the looped functions. It is much easier to do away with the loops.

From the command line you can see a list of machines that are in the machine queue by typing in lm:

```
Timbre: lm
   152D4 : processRxFrames
   15724 : sfpSpiMachine
   16C24 : serviceAsyncLinkMachine
   14150 : sampleAnalogIn
   11A18 : monitorIo
   13BF6 : monitorDacHealth
```

Time Machine

Sometimes it is necessary to synchronize functions with time. This example is a time machine or one that takes action at certain moments:

```
void sampleAnalogIn(void)
{
  if (checkTimeout(&analogInTo)) // check to see if a timeout has occurred
  {
       Cell time = (analogInSampleTime TO_MSEC) - sinceTimeout(&analogInTo);

       setTimeout(time, &analogInTo);
       measureAnalogInputs();
  }
  activate(sampleAnalogIn);
}
```

In this case a timeout is set and then checked for by the time machine. Once the period of time has passed, it executes its action, measureAnalogInputs, and then it repeats. The sinceTimeout() takes care of any jitter in the machine and on average the sample time will be equal to analogInSampleTime. analogInTo is simply a variable holding a time in the future that is checked against the current time. Since this is run as a machine and the resolution is in milliseconds, there will be a certain amount of jitter due to the other machines running and the accuracy of a millisecond. If precise timing is required, then one would attach the machine to a timer interrupt or the ADC might already have timed sampling built in. In which case the machine retrieves the samples and does something with them when it runs.

State Machine

A state machine is one that keeps state and has multiple behaviours depending on state. This one is used blink the status LED at two different rates. This state machine uses a timeout:

```
void blinkBlonk(void)
{
  static struct {void (*action)(void); Long time; int next; } state[] = {
       {greenOff, 100 TO_MILLISECONDS, 1},
       {greenOn,  100 TO_MILLISECONDS, 2}, // blink
       {greenOff, 100 TO_MILLISECONDS, 3},
       {greenOn,  700 TO_MILLISECONDS, 0}};// blonk
  static Timeout bto = {0};
  static int current = 0;

  if (checkTimeout(&bto))
  {
       state[current].action();
       setTimeout(state[current].time, &bto);
       current = state[current].next;
  }
```

```
}

void blinkBlonkMachine(void)
{
  blinkBlonk();
  activate(blinkBlonkMachine);
}
```

The LED state machine is contained within the function `blinkBlonk` while the invocation of that machine is done by the `blinkBlonkMachine`. The functions `greenOn` and `greenOff` merely set the IO line driving the LED to the correct level. Most of the time the machine is just checking for a timeout which involves a read and test which is very efficient. When an action is required, it is direct. This breaks up the actions into smaller segments which can interleave with other actions improving response time. This could be programmed to run from a hardware timer but the programming would be more complex and the resource might not be available. In a conventional RTOS with tasks and loops this would require the overhead of creating a task and scheduling it. For this simple function, this is a nice low overhead solution.

API

Machines are any C function without input or output. There are four functions to the API:

activate(machine) - queue up a machine to be run
deactivate(machine) - remove machine from queue
activateOnce(machine) - queue up only one instance
runMachines() - called at the end of the application startup code

CLI

The command line interface (CLI) is a machine which interprets input and provides the developer a window into the system. There is a data stack to hold values for calculations. Numbers are automatically pushed on the stack while operators take their arguments from the stack and leave results on the stack.

A compact list of words on the target can be displayed by typing in `words` at the command prompt:

```
Timbre: words
dbg  swap  dup  drop  >r  r  r>  sp!  over  ?dup  execute  and  or  xor  not  +
-  negate  /mod  /  mod  *  =  <  >  u<  u>  abs  max  min  @  !  c@  c!  s@
s!  +b  -b  l@  l!  cmove  fill  erase  here  allot  c,  ,  koff  kon  emit  cr
count  type  base  hex  decimal  hold  <#  #  #s  sign  #>  .r  .  ]  word
>b :  constant  variable  shift  .b  .d  .h  .s  dump  words  lm  start  end
time  help  [  (  literal  if  else  endif  begin  again  while  repeat  until
for  next  exit  '  "  ."  reveal  ;
```

The command prompt reflects the nature of the board. To get a more comprehensive list of the words in alphabetical order type in help. help will take an argument and return all matching words in the list.

```
Timbre: help help
help   print list of one liners about words in dictionary
```

To dump memory, type in the address in hex, followed by the number of (16 byte) lines to view, followed by dump:

```
Timbre: 0 4 dump
            00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000:   00 80 29 0C 00 00 11 34 00 00 0B CC 00 00 0B CC ..)....4........
00000010:   00 00 0B CC 00 00 0B CC 00 00 0B CC 00 00 0B CC ................
00000020:   00 00 0B CC 00 00 0B CC 00 00 0B CC 00 00 0B CC ................
00000030:   00 00 0B CC 00 00 0B CC 00 00 0B CC 00 00 0B CC ................
```

Numbers are typed in are decimal or hex if preceded by `0x` as in `0xabcd`. Values are placed on a data stack and are operated on from there with the results placed back on the stack. For instance `1 2 +` would first place 1 then 2 on the stack and then + would pop the two numbers, add them and leave 3 behind on the stack. In the same manner, arguments to words are prefixed. 0b and 0c can be used to prefix binary and octal numbers. Dot '.' is used to print out a number in the current radix. Doth '.h' and dotb '.b' are used to print out values in hex and binary. Values entered are pushed onto a stack of values which can be arguments to words or used for math. For example:

```
Timbre: 0c10 . 8
Timbre: 0b10 . 2
Timbre: 0x10 . 16
Timbre: 10 . 10
Timbre: 0c10 .h        8
Timbre: 0c10 .b  00001000
Timbre: 0b10 .h        2
Timbre: 0b10 .b  00000010
```

List of words

Words may be added to the dictionary by editing the file bootbindings.txt in the WordLists folder. This can be used to make C functions and variables available to the command line. At compile time, a Python script parses this file producing 3 output files: a C source and header file as well as a text file which lists all the words. This an example:

```
Word list for firmware  generated by parsewords.py  Feb 11, 2015  07:15:06

 [v] for variable
 [i] for compile words
 ( a - b ) stack before operation 'a' and after operation 'b'; right is top

!   ( n a - ) store next into memory using top as address (processor sized)
"   [i]  ( - s ) enclose text up to next " in a count prefixed string
#   ( n - n' ) convert a digit from n
#>   ( n - a c ) finish number sequence and return address and count
#s   ( n - 0 ) convert all digits in n
'   [i]  ( - a ) get execution address of following word
(   [i]  start of comment till end of line or )
*   ( n m - p ) multiply next data stack item by top and leave on top
+   ( n m - p ) add top two data stack items and leave on top
+b   ( b a - ) turn on b bits at address a: 0b10001 em +b
,   ( n - ) allocate 1 cell and put n into it
-   ( n m - p ) subtract top data stack item from next item and leave on top
-b   ( b a - ) turn off b bits at address a: 0b10001 em -b
```

. (n -) print n in current number base
." [i] print text up until next "
.b (n -) print n in binary
.d (n -) print n in decimal
.h (n -) print n in hex
.r (m n -) print m in right field of n digits
.s print number of items on data stack
/ (n m - q) divide next data stack item by top and leave on top
/mod (n m - q r) return divide and modulus from top item into next item
0stats reset machine stats
: start a macro definition
; [i] end a macro
< (n m - f) leave a boolean on stack indicating if next is less than top
<# inititiate a number sequence
= (n m - f) leave a boolean on stack after equating top two data stack items
> (n m - f) leave a boolean on stack indicating if next is greater than top
>r (n -) (R - n) push the top item of the data stack onto the return stack
?dup (n - n n | - 0) duplicate top data stack item if not 0
@ (a - n) return contents of memory using top stack item as the address (processor sized)
[[i] exit macro mode
] enter macro mode
abs (n - n|-n) top data stack item is made positive
again [i] end of a continuous loop construct
allot (n -) reserve n bytes after end of dictionary
and (n m - p) bitwise AND top two data stack items and leave on top
base (- a) return address of number radix
begin [i] start of a loop construct
c! (c a -) store next into memory using top as address (8 bit)
c, (c -) allocate and 1 byte and put value in it
c@ (a - c) return contents of memory using top stack item as the address (8 bit)
cmove (s d n -) move n bytes from s to d
constant (n -) give n a name
count (a - a' c) leave first character and incremented address on stack
cr send end of line to output device
dbg [v] set debug level 0 - quiet, 1 state changes, 2 traffic: 2 dbg c!
decimal interpret all subsequent numbers as decimal
drop (n -) throw away the top data stack item
dump (a n -) dump n 16-byte rows of memory starting at address a
dup (n - n n) make a copy of the top data stack item
else [i] otherwise part of an if statement
emit (c -) send c to output device
end display elapsed time since start
endif [i] end of else or if statement
erase (s n -) erase n bytes from s
execute (a -) use the top data stack item as a function call
exit [i] exit macro
fill (s n x -)fill n bytes from s with x
for [i] (n -) start of a loop which runs n times
help print list of one liners about words in dictionary

here (- a) return address of end of dictionary
hex interpret all following numbers as hex
hold (c -) hold a character in number sequence
if [i] (n -) execute following code if top of stack is non-zero
koff turn off automatic key echo
kon turn on automatic key echo
l! (n a -)store next into memory using top as address (processor sized)
l@ (a - n)return contents of memory using top stack item as the address (32 bit)
literal [i] (n -) enter a literal value into a macro
lm list machines
max (n m - n|m) leave maximum of top two stack items
min (n m - n|m) leave minimum of top two stack items
mod (n m - r) modulus next data stack item by top and leave on top
negate (n - -n) two's complement of top data stack item
next [i] end of a for loop
not (n - n') invert all bits on the top data stack item
or (n m - p) bitwise OR top two data stack items and leave on top
over (n m - n m n) copy 2nd data stack item to top of data stack
r (- n) (R n - n) copy the top item of the return stack onto the data stack
r> (- n) (R n -) move top item on return stack to data stack
repeat [i] go back to the begin part
reveal [i] allow macro to call itself
s! (h a -) store next into memory using top as address (16 bit)
s@ (a - h) return contents of memory using top stack item as the address (16 bit)
shift (n m - p) shift n by m bit left for minus and right for positive
sign (n -) prepend sign to number sequence if n is negative
sp! (... -) empty the data stack
start note the starting time
stats show time statistics on machine loop
swap (n m - m n) swap top two items on the data stack
time display local time
type (a n -) output n characters starting at a
u< (u v - f) leave a boolean on stack indicating if unsigned next is less than top
u> (u b - f) leave a boolean on stack indicating if unsigned next is greater than top
until [i] (n -) go back to the begin statement if stack is zero
variable (n -) give n a place to be stored at a name
version display application name and build date/time
while [i] (n -) conditional choice in a loop construct
word (c -) parse characters up to c from input to here
words list all words in dictionary
xor (n m - p) bitwise XOR top two data stack items and leave on top

Porting ET

Porting ET to run in different environment and micros is a matter of providing a timer that calls a function once a millisecond and an interface to a UART. The main function will initialize the necessary resources and then call runMachines in a forever loop.

Development

Once a basic system is up and running, then it is a matter of adding the application components. This is done in an interactive incremental manner, testing and adding parts as they are developed.